More General Optimal Offset Assignment

Sven Mallach

Institut für Informatik
Universität zu Köln, 50969 Köln, Germany

November 3, 2014

Abstract

We present exact approaches to the General Offset Assignment problem arising in the address code generation phase of compilers for application-specific processors. First, integer programming models for architecture-dependent and theoretically motivated special cases of the problem are established. Then, these models are extended to provide the first widely applicable formulations for the most general problem setting, supporting processors with several address registers and complex addressing capabilities. Existing heuristics are similarly extended and practical applicability of the proposed methods is demonstrated by experimental evaluation using an established and large-scale benchmark set. The experiments also allow us to study the impact of exploiting more complex memory addressing capabilities on the address computation costs of real-world programs. Further, we show how to integrate operand reordering techniques based on commutative instructions into existing solution approaches.

1 Introduction

We study the offset assignment problem arising in the address code generation phase of compilers for digital signal and other application-specific processors. To save silicon area, such processor designs often have narrow instruction widths limiting the number of bits available for memory addressing. Typically, there is no support for indirect addressing modes that combine a base address held in a register with an immediate offset to build the effective address of a memory operand (sometimes called *base plus offset* addressing). However, DSPs and other specialized Harvard architectures usually provide an address generation unit (AGU) supporting pointer arithmetic to be done in parallel to the main data path. The additional hardware can help to at least partially compensate the drawbacks if exploited properly. It supports instructions that permit to manipulate an address register (AR) in the same clock cycle as another instruction referencing it. Either, the respective modifications are encoded implicitly (effectively moving the encoding of the offset into the instruction opcode) or the instructions permit to add (subtract) values within a small architecture-dependent *auto-modify range* $[-r, r]$ to (from) the address held in the AR [1]. AR modifications by absolute values larger than r however need additional explicit address arithmetic instructions.

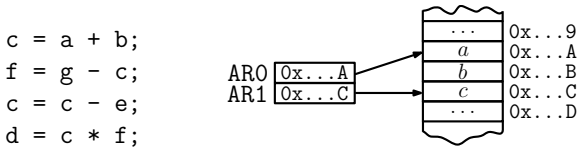

Figure 1: A sample code fragment and an illustration of ARs referencing memory locations of variables.

With these specialized instructions at hand, one can consider the complexity of indirect addressing to have been moved from hardware to software, relying on compilers in order to exploit the processor's capabilities for fast memory addressing of statically allocated variables. However, optimal exploitation asks for the optimal solution of two interdependent problems, namely to determine a stack memory layout of the variables accessed and to select an address register responsible for each of the accesses. Conversely, address computation overhead may result from two main issues. An inappropriate storage layout may necessitate additional explicit address arithmetic instructions for 'jumps' to addresses that have a distance larger than r. Further, if the processor provides multiple ARs, a poor choice of the ARs responsible for particular accesses may result in superfluous immediate AR loads and also unnecessary 'jumps'. Since address calculations make up a significant part of machine instructions, optimizing these decisions may considerably reduce the code size and speed up the program at the same time. Indeed, various experimental studies [2, 3, 1] show that optimized configurations lead to significant savings in practice.

During compilation, the instruction scheduling phase determines the *access sequence* to local stack variables. It can be extracted by simply concatenating the referenced variables of each three-address-code instruction $c = a\ op\ b$ in the order $a\ b\ c$. For example, the program fragment in the left of Fig. 1 refers to the variables $\mathcal{V} = \{a, b, c, d, e, f, g\}$ that are accessed in the order $S = a\ b\ c\ g\ c\ f\ c\ e\ c\ c\ f\ d$. Tab. 1 shows pseudo machine code for this program fragment and three potential stack layouts A, B, and C of $\mathcal{V}$. Layout A complies with the order of first use of the variables in S. On a processor with only a single AR, this layout would require six explicit address arithmetic instructions (`ADAR` and `SBAR`). An optimized layout (B) already reduces the necessary number of such instructions to three by increasing the use of autoin-/decrement instructions (with `*(ARx)+`/`*(ARx)-`). If the memory layout is optimized for a use of two ARs (C) and also an optimal AR assignment is computed, it becomes possible to cover the access sequence even without any explicit address arithmetic at all. Assuming the cost of an immediate AR load and the cost of an address arithmetic instruction to be both one, the optimal total cost with one AR is four, with two ARs it is two. Notably, layout A and B have no register assignment that leads to a total cost smaller than three with two or more ARs.

2 Related Work and Contribution

The *General Offset Assignment (GOA)* problem is defined for processors with k address registers and an auto-modify range of r. Given an access sequence

Instruction	AR0
LDAR AR0, &a	&a
LOAD *(AR0)+	&b
ADD *(AR0)+	&c
STOR *(AR0)+	&g
LOAD *(AR0)-	&c
SUB *(AR0)	
ADAR AR0,2	&f
STOR *(AR0)	
SBAR AR0,2	&c
LOAD *(AR0)	
ADAR AR0,3	&e
SUB *(AR0)	
SBAR AR0,3	&c
STOR *(AR0)	
ADAR AR0,2	&f
MUL *(AR0)	
ADAR AR0,2	&d
STOR *(AR0)	
$A = a\ b\ c\ g\ f\ e\ d$	

Instruction	AR0
LDAR AR0, &a	&a
LOAD *(AR0)+	&b
ADD *(AR0)	
ADAR AR0,2	&c
STOR *(AR0)-	&g
LOAD *(AR0)+	&c
SUB *(AR0)+	&f
STOR *(AR0)-	&c
LOAD *(AR0)	
ADAR AR0,3	&e
SUB *(AR0)	
SBAR AR0,3	&c
STOR *(AR0)+	&f
MUL *(AR0)+	&d
STOR *(AR0)	
$B = a\ b\ g\ c\ f\ d\ e$	

Instruction	AR0	AR1
LDAR AR0, &a	&a	
LOAD *(AR0)+	&b	
ADD *(AR0)+	&c	
STOR *(AR0)		
LDAR AR1, &g		&g
LOAD *(AR1)-		&e
SUB *(AR0)+	&f	
STOR *(AR0)-	&c	
LOAD *(AR0)		
SUB *(AR1)		
STOR *(AR0)+	&f	
MUL *(AR0)+	&d	
STOR *(AR0)		
$C = a\ b\ c\ f\ d\ e\ g$		

Table 1: Pseudo machine codes for the code fragment from Fig. 1 assuming different memory layouts A, B and C and either one (A and B) or two (C) available address registers.

$S = \{s_1, s_2, \ldots, s_{|S|}\}$ on program variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, it asks for a stack memory layout of the variables, i.e., a permutation $\pi : \mathcal{V} \to \{1, \ldots, n\}$ and an assignment $A : S \to \{1, \ldots, k\}$ of accesses to address registers that well exploits the auto-modify range r.

Most of the literature considers special cases of the problem, where either $r = 1$, $k = 1$ or both. For $k = 1$, the problem is called the *Simple Offset Assignment (SOA)* problem. It reduces to the task to find a stack memory layout that allows as many accesses as possible to be performed by auto-modify instructions on the single available address register. It was first considered by Bartley [4] in 1992. Assuming also $r = 1$, he proposed to model the variable relationships contained in an access sequence by an *access graph* $G = (V, E)$. The set of vertices V corresponds to the variables and there is an edge $e =$

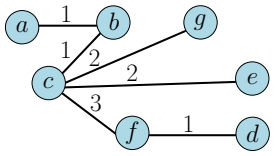

Figure 2: Access graph for the code fragment from Fig. 1.

$\{u, v\} \in E$ with weight $w(e)$ if the variables u and v appear subsequently in the access sequence for $w(e)$ times. Fig. 2 shows the access graph that corresponds to the sample code fragment from Fig. 1.

Bartley recognized a close relationship of SOA (with $r = 1$) to the Maximum Weight Hamiltonian Path (MWHP) problem and developed a first greedy heuristic to solve it. In subsequent research, Liao [5] showed that SOA is rather equivalent to a Maximum Weight Path Cover (MWPC) problem instead and gave a formal proof of its strong $\mathcal{NP}$-hardness. Based on these results, he proposed a simpler and faster heuristic producing solutions with the same quality as Bartley's and also a first exact Branch-and-Bound procedure. Since then,

many heuristics and also an optimal approach capable to solve larger instances have been developed and also been subject to experimental studies [2, 3, 1].

Since 2014, the situation is similar for the general variant with $k \geq 1$ and $r = 1$. Using the exact approach presented in [6], several GOA heuristics [7, 8, 9] could be evaluated in terms of quality relative to the optimum for the first time. In practice, GOA is often solved heuristically by first partitioning the set of program variables w.r.t. the available ARs and then solving a SOA problem for each of the ARs. This strategy allows to reuse available SOA algorithms but inherently constrains all accesses to a particular variable to be carried out by the same AR. This may preclude optimal results as is comprehensively discussed by Huynh et al. [2]. Even more, the results in [6] suggest not to partition the variables a priori, but to first compute a memory layout for them and an address register assignment afterwards.

In this paper, we highlight the key ideas and techniques that let the exact GOA approach for $k \geq 1$ and $r = 1$ presented in [6] be successful in solving a wide range of instances to optimality in reasonable time. The mentioned article also contains a correction of the only previous exact approach by Ozturk et al. [10] that was originally designed for arbitrary ranges r. However, as the experiments in [6] show, the method is not capable to solve larger instances, even for $r = 1$. It suffers from a quickly growing number of variables and constraints and does neither exploit the combinatorial structure nor symmetries inherent to the problem. Here, we develop an alternative formulation that takes advantages of the techniques presented in [6], yet generalizing to arbitrary auto-modify ranges and still remaining relatively moderate in size. Using this approach and by extending existing heuristics, we provide the first experiments for ranges larger than one on a larger set of instances and study the effect of exploiting larger auto-modify capabilities on the total address computation costs. Further, we address a commonly observed criticism associated with GOA, namely that it is not clear how it relates to operand reordering techniques such as, e.g, [11, 12, 13], which may also result in reduced address computation overhead. We present a method to integrate commutativity-based operand reordering into the address register assignment part of the optimization process. Combined with our optimal approaches for GOA, this allows to create globally optimal address code.

3 Optimal Address Register Assignment

3.1 A Minimum-Cost Flow Model

Suppose for now that a memory layout $\mathcal{L}$ of the program variables $\mathcal{V}$ has already been fixed and we are now asked to compute an optimal address register assignment (ARA) for k address registers w.r.t. $\mathcal{L}$ and the input access sequence S. For $r = 1$, an exact solution to this problem has been proposed by Gebotys [14, 15]. It is based on a minimum-cost circulation network that contains a vertex for each access in S and a directed arc for each pair of accesses u, v such that v succeeds u in S. In [6], it is shown how the circulation problem can be transformed into an equivalent minimum cost flow problem which we will now further describe.

Let V_S be an ordered set of vertices associated with the accesses in sequence

S. The network $N = (V_N, A)$ is obtained by setting $V_N = V_S \cup \{s, t\}$ and A to be the union of the arc sets $\{(s, v) \mid v \in V_S\}$, $\{(v, w) \mid v, w \in V_S, v < w\}$, $\{(v, t) \mid v \in V_S\}$. As a small example, let $\mathcal{V} = \{a, b, c, d\}$, $S = a\ d\ c\ c\ a\ b$ and assume $\mathcal{L} = d$ - a - c - b (which is optimal for $k \geq 2$ ARs). After adding artificial 'source' (s) and 'sink' (t) vertices, the associated minimum cost flow network looks as depicted in Fig. 3.

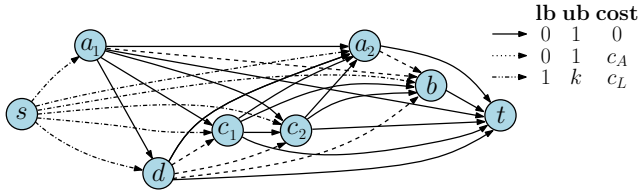

Figure 3: Minimum cost flow network for $S = a\ d\ c\ c\ a\ b$ assuming $\mathcal{L} = d$ - a - c - b.

The total flow leaving the source is restricted to be at least one unit and at most k units. All arcs have capacity one and each sequence vertex is constrained to receive and supply exactly one unit of flow. Hence, each unit of flow leaving s essentially delivers a path of accesses before it proceeds to t. The cost of an arc between two accesses is zero if and only if the two associated variables are equal or adjacent in $\mathcal{L}$. Otherwise the cost c_A of an address arithmetic instruction is associated with the arc (drawn dashed in Fig. 3). The cost of an immediate AR load is installed on every s-leaving arc and all costs of arcs entering t are zero. So if the selection of paths through the network is based on a min-cost criterion, then each of the resulting paths can be interpreted as an optimal series of accesses performed using a distinct AR. An optimal solution (assuming $c_L = c_A$) to the example from Fig. 3 for $k \geq 2$ ARs is depicted in Fig. 4.

Let $y_{u,v}$ be a flow variable for each arc $(u, v) \in A$ and $c_{u,v}$ its associated cost. The LP formulation corresponding to the described min-cost flow problem is then:

$$
\begin{array}{lrcl}
\min & \sum_{(u,v)\in A} c_{u,v} y_{u,v} & & \\
s.t. & \sum_{(v,w)\in A} y_{v,w} & = 1 & \text{for all } v \in V_S \\
 & \sum_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } v \in V_S \\
 & \sum_{v\in V_S} y_{s,v} & \leq k & \\
0 \leq & y_{u,v} & \leq 1 & \text{for all } (u, v) \in A
\end{array}
$$

The restriction of the in- and out-degrees of all sequence-vertices to one by the second and third constraints highlights the aforementioned 'path selection' property of this model. Actually, they make the usual flow conservation constraints of network flow problems obsolete; any unit of flow sent from s to satisfy the equations must finally arrive at t. Further, lower bounds on the flow on out- (in-) arcs of the source (target) are not necessary since the former (latter) must be satisfied due to the in- (out-) degree equation of the first (last) access vertex. All flow variables may be only zero or one and combinatorial algorithms

as well as linear programming can be used in order to obtain integral solutions in polynomial time. Optimal LP solutions will always be integral due to the unimodularity property of the associated constraint matrix [16].

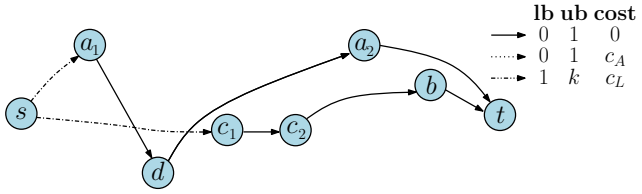

Figure 4: An optimal solution to the example from Fig. 3.

3.2 Taking Commutativity Into Account

Sometimes, address computation overhead can also be avoided by simply reordering the accesses to operands where this is possible due to the commutativity of the respective operations. For example, assume that variable b was last accessed and `c = a + b` is the next instruction. Then it is typically beneficial to access b first if a and b are not adjacent in the stack layout.

Reordering opportunities can be incorporated into the general approach by Gebotys and also into the just described flow model. If u and v are the two respective operands belonging to a commutative instruction, one can replace the corresponding flow arc $(u, v) \in A$ by a flow *edge* $\{u, v\}$ that can be used in both directions. Since we may reasonably assume to have three-address-code instructions only, the model guarantees that each vertex of the network has at most one neighbor that is adjacent by an edge rather than by an arc. At these particular vertex pairs, the flow in the network is then permitted to also move 'backward' while the constraints that each vertex must have exactly one incoming and outgoing unit of flow will preserve the overall feasibility of the problem and correctness of the solutions.

Suppose the access sequence $S = a\ d\ c\ c\ a\ b$ from the previous subsection is stemming from the computations `c = a * d; b = c * a`. Then both operations are commutative and the arcs (a_1, d) and (c_2, a_2) in the flow network become edges in the proposed methodology. Indeed, sending flow 'backward' along these edges allows for a better solution with cost only one as is depicted in Fig. 5.

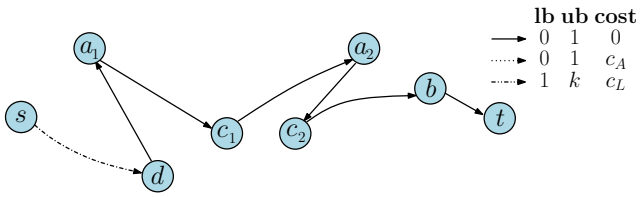

Figure 5: An optimal solution to the example from Fig. 3 exploiting commutativity.

4 Optimal General Offset Assignment

To solve GOA to global optimality, we need to solve the two interdependent problems in an integrated fashion, i.e., we need to find a memory layout that will allow us to create the best possible address register assignment.

A key observation is that the objective function is the only point where the memory layout influences the concrete ARA network problem to be solved. The cost of an access transition (u, v) in the network described in Sect. 3 is zero if and only if the variables associated with u and v are equal or neighbors in the memory layout. Otherwise, a positive cost c_A reflecting the overhead of an additional address arithmetic instruction is assigned. Moreover, there is no reason to not redefine this rule for $r > 1$, i.e., to assign arcs $(u, v) \in A$ the cost zero if u and v are no more than r positions apart from each other and c_A otherwise.

However, in terms of modeling the feasible solutions of GOA problems, it makes a considerable difference whether r is equal to one or may be larger. We will now first discuss the approach for $r = 1$ from [6] and then proceed to the more general case.

4.1 GOA with $r = 1$

If $r = 1$, then an access transition can only be realized with autoin-/decrement instructions if the two associated variables are either identical or neighbors in the memory layout.

Clearly, a memory layout is basically a permutation of the variables. As already indicated in Sect. 2, possible permutations of the variables $\mathcal{V}$ can be modeled by the Hamiltonian paths of a complete undirected graph G with vertex set $\mathcal{V}$. From an integer programming point of view, it is easier to model Hamiltonian cycles instead of paths. By appending an artificial vertex z to the graph, i.e., setting $G' = (V, E)$ with $V = \mathcal{V} \cup \{z\}$, we are able to derive a unique Hamiltonian Path P in G from each Hamiltonian Cycle C in G' by simply removing the vertex z and its two adjacent edges in C [3, 6].

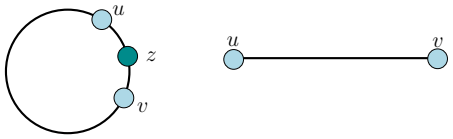

Figure 6: Extracting a Hamiltonian path from a Hamiltonian cycle.

To model GOA completely, we will now always consider two graphs. Firstly, a complete graph $G = (V, E)$ where $V = \mathcal{V} \cup \{z\}$ as just described. Secondly, we have a network $N = (V_N, A)$ with $V_N = V_S \cup \{s, t\}$ where V_S is a vertex set related to the accesses contained in the input sequence S, just like in Sect. 3. Let $A_S = \{(v, w) \mid v, w \in V_S, v < w\}$ and $A = A_S \cup \{(s, v) \mid v \in V_S\} \cup \{(v, t) \mid v \in V_S\}$. Since all access vertices in V_S are instances of the program variables represented by the vertices $\mathcal{V}$, we may define a corresponding unique mapping $\sigma : V_S \to \mathcal{V}$. For ease of reference, we further split the set A_S into $A_S^{\neq} = \{(u, v) \in A_S,\ \sigma(u) \neq \sigma(v)\}$, i.e., the set of arcs between accesses that do not refer to the same associated program variable and, analogously, the set $A_S^{=}$.

In addition to the flow arc variables $y_{u,v}$ for each arc $(u,v) \in A$, we associate edge decision variables $x_{u,v} \in \{0,1\}$ with the edges $\{u,v\} \in E$ that have no associated costs. The variable $x_{u,v}$ is equal to one if the edge $\{u,v\}$ is part of the computed Hamiltonian cycle (u and v are neighbors in the memory layout), and zero otherwise. Since G is undirected, the variables $x_{u,v}$ are only defined for $u < v$. Slightly disregarding mathematical precision, we write $x_{\sigma(u),\sigma(v)}$ when referring to the associated edge decision variable of $y_{u,v}, (u,v) \in A_S^{\neq}$, irrespective of whether $\sigma(u) < \sigma(v)$ or $\sigma(u) > \sigma(v)$. Exploiting these variable relationships, we can express the cost of an access transition $y_{u,v}$ with $(u,v) \in A_S^{\neq}$ by $(1 - x_{\sigma(u),\sigma(v)})c_A$ while the cost of each variable $y_{u,v}$ for $(u,v) \in A_S^{=}$ is zero. This leads to a first quadratic integer programming formulation for GOA.

4.1.1 Quadratic Formulation

$$
\begin{aligned}
\min \quad & \sum_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v} + \sum_{v \in V_S} c_L y_{s,v} \\
s.t. \quad & \sum_{\{u,v\}\in E} x_{u,v} && = 2 && \text{for all } v \in V \\
& x(E(W)) && \leq |W| - 1 && \text{for all } \emptyset \neq W \subsetneq V \\
& \sum_{(u,v)\in A} y_{u,v} && = 1 && \text{for all } u \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} && = 1 && \text{for all } v \in V_S \\
& \sum_{v\in V_S} y_{s,v} && \leq k \\
& x_{u,v} && \in \{0,1\} && \text{for all } (u,v) \in E \\
& y_{u,v} && \in \{0,1\} && \text{for all } (u,v) \in A
\end{aligned}
$$

This integer program is essentially the min-cost flow formulation from Sect. 3 appended by inequalities enforcing the x-variables to correspond to a Hamiltonian cycle of G and with a new objective function linking the two subproblems. The objective function simply sums up the terms $(1 - x_{\sigma(u),\sigma(v)})c_A$ for all arcs $(u,v) \in A_S^{\neq}$ and all costs $\sum_{v\in V_S} c_L y_{s,v}$ for initial address register loads.

The set of feasible Hamiltonian cycles of G can be expressed using the standard formulation of the Traveling Salesman Problem (TSP): The additional equations force any vertex to be adjacent to exactly two other vertices. The subsequent inequalities are called *subtour elimination constraints* [17]. Here, $E(W) = \{\{u,v\} \in E \mid u,v \in W\}$ for any $W \subseteq V$ such that the inequalities exclude solutions containing any cycle w.r.t. the vertex sets W and $V \setminus W$ from the feasible set.

The above integer program is quadratic in its objective function. We may linearize it using the standard linearization approach. However, first we simplify. The term

$$\min \sum_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v}$$

can also be written as

$$\min \Big(\sum_{(u,v)\in A_S^{\neq}} y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} x_{\sigma(u),\sigma(v)} y_{u,v}\Big) c_A.$$

We then need $|A_S^{\neq}|$ new variables $z_{u,v} = x_{\sigma(u),\sigma(v)} y_{u,v}$ and three linearization constraints for each of the new variables:

$$\begin{aligned} z_{u,v} &\leq x_{\sigma(u),\sigma(v)} \\ z_{u,v} &\leq y_{u,v} \\ z_{u,v} &\geq x_{\sigma(u),\sigma(v)} + y_{u,v} - 1 \end{aligned}$$

After this transformation, the objective function becomes:

$$\min \sum_{(u,v)\in A_S^{\neq}} c_A y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} c_A z_{u,v} + \sum_{v\in V_S} c_L y_{s,v}$$

Clearly, the number of product variables to be introduced, $|A_S^{\neq}|$, must be strictly smaller than the total number of flow arc variables which is $(|S|+2)\cdot(|S|+1)$. Hence, in total the linearized version of the above formulation has strictly less than $|\mathcal{V}|\cdot(|\mathcal{V}|-1)/2 + ((|S|+2)\cdot(|S|+1))$, i.e., $\mathcal{O}(|\mathcal{V}|^2 + |S|^2)$ variables. The number of subtour elimination constraints is exponential in $|\mathcal{V}|$ such that it is preferable to not consider all of them in the solution process from the beginning, but to separate them instead as we will describe more detailed in Sect. 5.1. The number of remaining constraints is strictly less than $1 + |\mathcal{V}| + 2(|S|+2) + 3(|S|+2\cdot(|S|+1))/2$, i.e., $\mathcal{O}(|\mathcal{V}| + |S|^2)$.

Remarkably, all these numbers are independent from the number k of ARs available. The more access pairs in S refer to the same variable, the less product variables and associated constraints are needed.

4.1.2 Linear Formulation

By further inspection and exploiting the fact that there are only two cases for each arc $(u,v) \in A_S^{\neq}$, namely that it either has the assigned cost c_A or assigned cost zero, we found a way to linearize the problem inherently, that is without generating any products that need a subsequent linearization. The main idea is to replace every variable (arc) between two accesses $y_{u,v}, (u,v) \in A_S^{\neq}$ by two new variables (arcs) $y_{u,v}^0$ and $y_{u,v}^1$ reflecting the two mentioned cases (cf. Fig. 7). The set $A_S^{\neq}$ is therefore further split into the corresponding new arc sets A_S^0 and A_S^1. For every arc $(u,v) \in A_S^{=}$, we keep the former variable $y_{u,v}$ with zero cost as before. We also skip the superscript when referring to flow variables disregarding their costs or if only one instance exists.

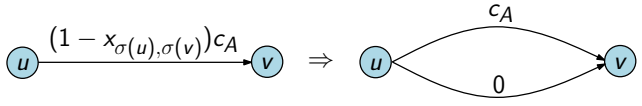

Figure 7: Replacing arcs with dynamic costs by two arcs with static costs.

The new network N has now the arc set $A = A_S^0 \cup A_S^1 \cup A_S^{=} \cup \{(s,v) \mid v \in V_S\} \cup \{(v,t) \mid v \in V_S\}$. The new objective is of course to minimize the selected arcs with positive costs assigned. This is a linear expression in the set of variables. However, we now have to restrict the use of zero-cost arcs. As before in the quadratic model, it should only be possible to use them if the respective corresponding variables are neighbors in the access sequence. With

the newly introduced variables this can easily be enforced using the following constraints:

$$y^0_{u,v} \quad \leq x_{\sigma(u)\sigma(v)} \qquad \text{for all } (u,v) \in A^0_S$$

Further, the following constraint is valid for the model:

$$y^1_{u,v} \quad \leq 1 - x_{\sigma(u)\sigma(v)} \qquad \text{for all } (u,v) \in A^1_S$$

However, since zero-cost arcs are preferred by the objective function, there will never be a variable $y^1_{u,v} = 1$ in an optimum solution where also $x_{\sigma(u)\sigma(v)} = 1$, even if these constraints are not present. Therefore, this constraint will also have only marginal impact on the solution process and can be omitted.

A complete linear IP formulation for GOA with $r = 1$ is then:

$$\begin{array}{lll}
\min & \sum_{(u,v)\in A^1_S} c_A y^1_{u,v} + \sum_{v\in V_S} c_L y_{s,v} & \\
s.t. & \sum_{\{u,v\}\in E} x_{u,v} = 2 & \text{for all } v \in V \\
& x(E(W)) \leq |W| - 1 & \text{for all } \emptyset \neq W \subsetneq V \\
& \sum_{(u,v)\in A} y_{u,v} = 1 & \text{for all } u \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} = 1 & \text{for all } v \in V_S \\
& \sum_{v\in V_S} y_{s,v} \leq k & \\
& y^0_{u,v} \leq x_{\sigma(u)\sigma(v)} & \text{for all } (u,v) \in A^0_S \\
& x_{u,v} \in \{0,1\} & \text{for all } (u,v) \in E \\
& y_{u,v} \in \{0,1\} & \text{for all } (u,v) \in A
\end{array}$$

The number of variables is the same as in the quadratic formulation since essentially for each $(u,v) \in A^{\neq}_S$ the product variable $z_{u,v}$ is replaced by a second flow arc variable $y^1_{u,v}$. However, $2|A^{\neq}_S|$ less constraints are needed.

4.2 GOA with $r \geq 1$

If the processor supports offset ranges r strictly larger than one, the mathematical modeling of the associated optimization problem becomes much more complicated. The Hamiltonian cycle method becomes inappropriate, since simple vertex adjacencies are not anymore sufficient in order to decide on the costs of access transitions in the flow network. Rather, the question whether an access transition $u \to v$ can be done without cost corresponds to the question 'Does there exist a path between u and v of length smaller or equal to r?' which cannot be answered by linear expressions in the x-variables of the preceding subsection. Even if we think of variables p_v expressing the precise position of each variable $v \in \mathcal{V}$ in the memory layout, the situation is not much better. The cost of an access transition $u \to v$ would then be zero if $|p_v - p_u| \leq r$. However, the cost function is not continuous such that, e.g., the expression $c_{u,v} \geq |p_v - p_u| - r$ would be incorrect as soon as $|p_v - p_u| - r > 1$.

The only straightforward way to model the problem correctly appears to be via using variables that encode the position of a variables explicitly. This can be done based on the famous *assignment problem* [18], in a similar way as it was also proposed by Wess and Gotschlich [19] for $k = 1$ and a slightly different problem setting. The model by Ozturk et al. [10] has also an assignment-based

character. However, as discussed in [6], their formulation was flawed and is not applicable to a wider range of instances. We will now develop a different formulation that has considerably less variables and constraints while preserving the advantages of our model for $r = 1$.

In the assignment problem, we have variables $x_{i,p}$ that take value one if item i is placed at position p and zero otherwise. To model the stack memory layout for $n = |\mathcal{V}|$ variables, we need exactly n^2 variables, since any variable may be placed at any position $p \in P, P = \{1, \ldots, n\}$. Clearly, every variable $v \in \mathcal{V}$ must be assigned exactly one position $p \in P$ and each position $p \in P$ must be assigned exactly one variable $v \in \mathcal{V}$. Hence, the corresponding constraints of the assignment problem are:

$$\begin{aligned} \sum_{p \in P} x_{v,p} &= 1 \qquad \text{for all } v \in \mathcal{V} \\ \sum_{v \in \mathcal{V}} x_{v,p} &= 1 \qquad \text{for all } p \in P \end{aligned}$$

Let $x_{u,a} = 1$ and $x_{v,b} = 1$ with $u \neq v$ and $a \neq b$. Then, an access transition $u \to v$ has cost zero if and only if $|b - a| \leq r$. For the following discussion, we introduce auxiliary variables $r_{u,v}$ for each pair of different variables $u, v \in \mathcal{V}$, expressing whether u and v are placed within range r or not. We will however not need these variables for the subsequently developed integer program. With the auxiliary variables at hand, we may express the following constraints:

$$\begin{aligned} r_{u,v} &\geq x_{u,b} + x_{v,a} - 1 \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| \leq r \\ r_{u,v} &\geq x_{u,a} + x_{v,b} - 1 \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| \leq r \end{aligned}$$

The constraints force $r_{u,v}$ to become one as soon as the assignment of positions to u and v is such that their distance is at most r. Further, they never enforce $r_{u,v}$ to be greater than one. We may also directly combine the constraints as follows:

$$r_{u,v} \geq \underbrace{x_{u,b} + x_{u,a}}_{\leq 1} + \underbrace{x_{v,a} + x_{v,b}}_{\leq 1} - 1 \text{ for all } u < v \in \mathcal{V} \text{ ; } a < b \text{ s.t. } |b-a| \leq r \quad (1)$$

On the other hand, we must make sure that $r_{u,v}$ is never assigned value one if the two variables are not placed within range r using the following constraints:

$$r_{u,v} \leq 2 - x_{u,b} - x_{u,a} - x_{v,a} - x_{v,b} \text{ for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| > r \quad (2)$$

In total, this would already amount to $\mathcal{O}(n^4)$ constraints, having not yet formulated constraints that restrict the use of the zero-cost flow-arc variables. However, we can still improve on that. First of all, for the same reason as in the linear IP presented in Sect. 4.1.2, we do not need to care about the positive cases that permit to use a flow arc variable $y^0_{u,v}$ but only forbid those cases where the use of $y^0_{u,v}$ is prohibited. Hence, we can completely omit the constraints (1). Now, we take a closer look on constraints (2) again, keeping in mind that each variable $v \in \mathcal{V}$ can be assigned at most one position from any strict subset Q of P, i.e., $\sum_{p \in Q} x_{v,p} \leq 1$ for all $v \in \mathcal{V}$ and $Q \subset P$. Say variable u is fixed at position a, then all the positions of v that make the transition $u \to v$ have non-zero cost

are the positions $p \in [1, a-r-1]$ and $p \in [a+r+1, n]$. Hence, by fixing one position, we can reformulate (2) by:

$$r_{u,v} \leq 2 - x_{u,a} - \underbrace{\sum_{p=1}^{a-r-1} x_{v,p}}_{\leq 1} - \underbrace{\sum_{p=a+r+1}^{n} x_{v,p}}_{\leq 1} \quad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| > r$$

Since we now exactly characterized under which conditions two variables u and v are not within range r, we can again exploit this to apply the correct restriction on the use of each zero-cost flow-arc variable $y^0_{u,v}$.

$$y^0_{u,v} \leq 2 - x_{u,a} - \sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} \qquad \text{for all } (u,v) \in A^0_S \text{ and } a \in P \quad (3)$$

Since, for any of the at most $\binom{|S|}{2}$ variables $y^0_{u,v}$, there are at most n positions where u can be fixed at, we obtain only $\mathcal{O}(|S|^2 \cdot |\mathcal{V}|)$ constraints (3) in total. The full IP formulation is then:

$$\begin{aligned}
\min \quad & \sum_{(u,v)\in A^1_S} c_A y^1_{u,v} + \sum_{v \in V_S} c_L y_{s,v} & & \\
s.t. \quad & \sum_{p \in P} x_{v,p} = 1 & & \text{for all } v \in \mathcal{V} \\
& \sum_{v \in \mathcal{V}} x_{v,p} = 1 & & \text{for all } p \in P \\
& \sum_{(u,v)\in A} y_{u,v} = 1 & & \text{for all } u \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} = 1 & & \text{for all } v \in V_S \\
& \sum_{v \in V_S} y_{s,v} \leq k & & \\
& y^0_{u,v} \leq 2 - x_{u,a} - \sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} & & \text{for all } (u,v) \in A^0_S \text{ and } a \in P \\
& x_{v,p} \in \{0,1\} & & \text{for all } v \in \mathcal{V} \text{ and } p \in P \\
& y_{u,v} \in \{0,1\} & & \text{for all } (u,v) \in A
\end{aligned}$$

Being more precise, the model has $|\mathcal{V}|^2 + (|S|+2) \cdot (|S|+1)$, i.e., again $\mathcal{O}(|\mathcal{V}|^2 + |S|^2)$ variables. The number of constraints is bounded from above by $1 + 2|\mathcal{V}| + 2(|S|+2) + ((|S|+2) \cdot (|S|+1)/2) \cdot |\mathcal{V}|$, i.e., $\mathcal{O}(|\mathcal{V}| \cdot |S|^2)$. In contrast to that, the formulation by Ozturk et al. has $\mathcal{O}(k \cdot |\mathcal{V}| \cdot |S| + |\mathcal{V}|^2)$ variables and $\mathcal{O}(|\mathcal{V}|^3 + k \cdot |S| \cdot |\mathcal{V}|^2)$ constraints (with larger constants).

5 Algorithms

5.1 Exact Branch-and-Cut Algorithms

Branch-and-Cut algorithms are similar to conventional LP-based Branch-and-Bound methods with the main difference that, at each Branch-and-Bound node, several LPs may be solved before a branching step is carried out. After solving an LP, it is tested whether the LP solution violates previously neglected or additional valid inequalities. If this is the case, these inequalities are added to

the LP as so-called 'cutting planes' and it is solved again. They ('separate') or 'cut off' the respective LP solution - it will not be a feasible solution of the LP solved next. The addition of inequalities may also improve the lower bounds on the objective function value provided by the LP solutions. Hence, they are essential, e.g., in proving optimality of a known solution. Either if no further violated inequality can be found or after some predefined number of iterations, a branching step takes place if the LP solution is still non-integral.

Cutting plane approaches are suitable especially if the number of constraints of an integer program is too large to be completely applied from the beginning, but a test for violation of these constraints can be done efficiently. This is also the case for the problem formulations presented in this article. The GOA formulations for $r = 1$ from Sect. 4.1 contain the subtour elimination constraints (SECs) whose number is exponential in the number of program variables $\mathcal{V}$. However, the associated separation problem can be solved in polynomial time using minimum cut algorithms [20]. This is exploited in our implementation which we will refer to as `GOA-IP`. The algorithm has an additional exact and polynomial-time separation procedure for Two-Matching-inequalities [21]. Violation of these is tested whenever a fractional LP solution did not violate any SEC. We also decided to separate inequalities (3) from Sect. 4.2 in the assignment-based solver implementation, subsequently named `GOA-AIP`. Although their number is polynomial in the size of the input data, these inequalities quickly become a limitation for larger instances due to increased LP solution times. Moreover, typically only a fraction of them is in fact required (i.e., ever violated) during the solution process.

We implemented `GOA-IP` and `GOA-AIP` using CPLEX 12.6 [22]. We disabled internal presolving techniques that are not compatible with the application of cutting planes but adopted all other default parameters. The optimization starts by relaxing integrality and the respective classes of inequalities that shall be separated as just described. In addition to the omitted inequalities, CPLEX may decide to separate further general cutting planes for integer programs and also the selection of the variables to branch on is left to CPLEX.

The LP solutions obtained during the solution process will typically not be integral. Here, *primal heuristics* play an important role. Their purpose is to construct good feasible solutions by exploiting the current LP solution, improving the upper bound on the optimal objective function value. They follow the general idea that variables with an LP value close to one are likely to be part of a good or even optimal solution. In particular, we greedily construct a memory layout as is indicated by the subsequent pseudocode. In case of `GOA-IP`, we select feasible edges $\{u, v\}$ in non-increasing order of the LP values of their corresponding variables $x_{u,v}$. In `GOA-AIP`, we assign each program variable $v \in \mathcal{V}$ the position that is mostly preferred by its assignment variables $x_{v,p}, p \in 1, \ldots, n$. Then, the network flow problem from Sect. 3.1 is solved to find an optimal ARA.

```
1: function primalHeuristic(x, N = (V_N, A))
2:     OA ← ComputeOffsetAssignment(x)
3:     SetCosts(N, OA)                    # set arc costs based on distances in OA
4:     ARA ← MinCostFlow(N)
```

```
1: function computeOffsetAssignment(x)                          # GOA-IP version
2:     G = (V, E') ← Graph with E' being the edges with at least one corresp. access
       transition in N
```

```
    SORT(E', x)                    # Sort edges non-increasingly w.r.t. their LP values
    INITIALIZEUNIONFIND(V)
    n ← |V|, m ← |E'|
    select ← ∅, count ← 0
    for i = 1 → n do
        deg(i) ← 0                                  # Initialize degrees to zero
    for i = 1 → m do
        e = {u, v} ← E'[i]
        if deg(u) < 2, deg(v) < 2, count < n and (FIND(u) ≠ FIND(v) or count =
    n − 1) then
            select ← select ∪ {e}, count ← count + 1
            deg(u) ← deg(u) + 1, deg(v) ← deg(v) + 1
            UNION(u, v)
    OA ← CONCATENATE(select)          # Join edges at common nodes, append
    isolated nodes
    return OA
function COMPUTEOFFSETASSIGNMENT(x)                          # GOA-AIP version
    n ← |V|
    for p = 1 → n do
        pos[p] ← free # Initialize all positions p ∈ P = {1, . . . , n} to be unassigned
    X_max ← Array of LP-values max{x_{v,p} | p ∈ {1, . . . , n}} for each v ∈ V
    SORT(V, X_max)                  # Sort variables V non-increasingly w.r.t. X_max
    for i = 1 → n do
        v ← V[i]
        X_v ← Array of LP-values x_{v,p} for each p ∈ {1, . . . , n}.
        SORT(P, X_v)                # Sort positions p ∈ P non-increasingly w.r.t X_v
        for j = 1 → n do
            p ← P[j]
            if pos[p] is free then
                pos[p] = v
                OA[v] = p
    return OA
```

5.2 Heuristics

For $r = 1$, the predominant heuristic strategy to solve GOA was to first partition the set of variables w.r.t. the available number of ARs, and to call a SOA algorithm to compute memory sublayouts for each of the partitions afterwards. Computational experiments [6] reveal that it is typically more suggestive to first set up a full memory layout and to compute then an optimal ARA based on that layout. For $r > 1$, this appears to be even more advisable since most of the existing SOA algorithms used as subroutines are designed for $r = 1$ (they iteratively select edges [1, 3]) and it is not trivial to generalize them. In contrast to that, Gebotys' network approach is easy to adapt for arbitrary auto-modify ranges as already discussed in Sect. 4 and also exploited by our primal heuristics.

Like in [6], we combine the optimal ARA approach with two simple strategies to compute memory layouts. The simplest version constructs a memory layout that corresponds to the order of first use of the program variables. The algorithm will be referred to as `GOA-OFU-MCF`. The second algorithm uses the most successful SOA heuristic `SOA-INC-TB` in [1, 3] to create a memory layout. It iteratively selects edges $\{u, v\} \in E$ from an access graph $G = (V, E)$ as it

has been exemplified in Sect. 2. The combination of this algorithm with the min-cost-flow based ARA computation is called `GOA-ITB-MCF`.

5.3 Min-Cost Flow Implementation

For all minimum cost flow computations (by the primal heuristics used in the exact solvers as well as by `GOA-OFU-MCF` and `GOA-ITB-MCF`), we called the network simplex algorithm provided by the LEMON C++ library [23] in version 1.3. The asymptotic running time of the network simplex algorithm strongly depends on the used pivoting rule. We relied on the default block search rule of the library [24] that has a worst-case time bound of $\mathcal{O}(nm^2)$ with $n = |V_N|$ and $m = |A|$ (since we have capacities and costs only zero and one), but typically performs much better in practice.

6 Experimental Evaluation

6.1 Setup and Test System

For our experimental evaluation, we use the OffsetStone benchmark set that has been extracted from 31 real-world application codes written in ANSI C. Among them are computationally intensive programs (e.g., audio, video and image compression, Fourier transformation) as well as control-dominated applications (e.g., gzip). It has been frequently used in publications dealing with offset assignment and therefore allows for meaningful comparisons. For details on how the instances were extracted, we refer to the original paper [1].

Being able to compute optimal solutions for various combinations of available address registers k and auto-modify ranges r on a larger set of instances for the first time, we evaluate the effect of varying k, r or both on the total offset assignment costs as well as on the quality of heuristic solutions. We assume the costs for address arithmetic instructions c_A and immediate AR loads c_R to be both equal to one and perform the experiments once for one, two, four and eight ARs and auto-modify ranges one, three and seven. We considered all instances that consist of at least three program variables, these are 2785 in total.

Our experiments were run single-threaded with an Intel Core $i7\text{-}3770T$ processor (2.5 GHz) on a Debian Linux system with 8 GB RAM, `g++` 4.7.2 and optimization level `-O2`. We measure the address computation overhead and average solution CPU times of five runs.

6.2 Results

Since extensive evaluation w.r.t. the quality of heuristics for $r = 1$ has been presented already in [6], we only briefly discuss the results for this case. The performance of `GOA-IP` only marginally differs from the mentioned reference since we used a more recent version of CPLEX. The number of instances that timed out after ten seconds is shown in the left of Tab. 2. The timeouts for $k = 1$ are only shown for the sake of completeness. Since this is the SOA case, the min-cost flow part is not needed and the problem can be solved instead by assigning static cost coefficients to the variables of the Hamiltonian cycle part of the problem. It would therefore be more suggestive to use the exact algorithm from [3].

In Tab. 2, we also see that GOA-AIP is not at all competitive to GOA-IP for $r = 1$. This was expectable; the primal heuristics work well, but for many instances the basic constraints of the assignment problem part do not suffice in order to obtain lower bounds that prove optimality of known solutions. However, with increasing auto-modify ranges, this effect more and more diminishes which can be similarly explained. With increasing r, the concrete memory layout becomes less important for an optimal address register assignment since more arcs in the min-cost flow network can be used without cost in any case. This effect is even stronger if the access sequence lengths are rather small which is often the case in OffsetStone as we will further discuss below. Hence, the lower bounds obtained from the LP and the upper bounds obtained by solutions found by the primal heuristics are much closer to each other and optimality of the latter can be proven much more often.

GOA-IP	$r = 1$
$k = 1$	11 (0.40%)
$k = 2$	34 (1.22%)
$k = 4$	58 (2.08%)
$k = 8$	55 (1.98%)

GOA-AIP	$r = 1$	$r = 3$	$r = 7$
$k = 1$	1253 (44.99%)	924 (33.18%)	472 (16.95%)
$k = 2$	1245 (44.70%)	880 (31.60%)	471 (16.91%)
$k = 4$	1246 (44.74%)	888 (31.89%)	477 (17.13%)
$k = 8$	1252 (44.96%)	881 (31.63%)	477 (17.13%)

Table 2: Number of instances timed out by the exact solvers after ten seconds.

Another expected result is that the property of GOA-IP to be relatively insensitive to the number k of ARs is inherited by GOA-AIP, not only since their numbers of variables and constraints are independent from k. Even for $r = 1$, GOA-AIP solves already $10-20\%$ more instances than the implementation of the fixed approach by Ozturk et al. in [6] and it never failed to solve a problem due to memory limitations which was often the case for the latter. We identified 1480 instances with up to 200 program variables that GOA-AIP could solve to optimality for all tested choices of k and r within the time limit of ten seconds. If an exact solver for arbitrary r is desired for practical application, GOA-AIP could be improved by several means, e.g., by adding additional cutting planes that are valid for (quadratic) assignment problems. In general, the quadratic nature of the problem and its interdependent structure of two subproblems suggests a reformulation as a semidefinite program or the application of Bender's decomposition approach. The latter is even more suggestive due to the fact that the min-cost-flow subproblem is polynomial-time solvable. In this sense, GOA-AIP is to be seen as a 'proof of concept' to produce first results for $r > 1$ that allow to evaluate heuristics and the effect of exploitation of larger auto-modify ranges on the quality of address code generation.

Investing more computation time, we derived optimal solutions for 1918 of the 2785 instances for all mentioned combinations of r and k. The OffsetStone instances are such that there are usually multiple access sequences associated with one program variable set. Hence, multiple min-cost flow problems need to be solved per instance, all referring to the same memory layout. Further, sequences may refer to disjoint subsets of the program variables such that the instances can be decomposed. The left image of Fig. 8 shows cumulated access lengths for the 1918 instances. The right one gives a fine-grain distribution of single access sequence lengths after decomposition. Unfortunately, the number of longer access sequences is rather small which is one of the already addressed

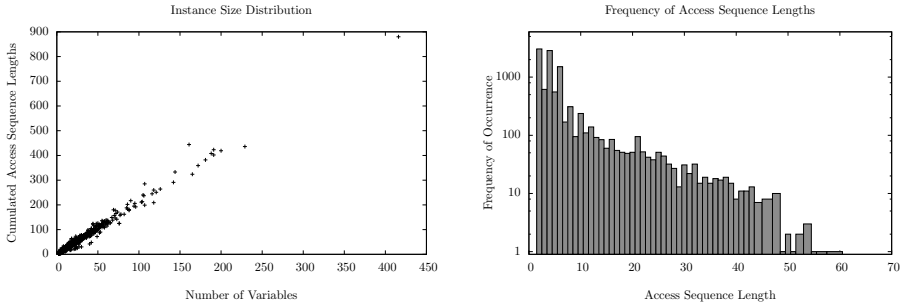

Figure 8: Distribution of the number of variables and access lengths across the instances.

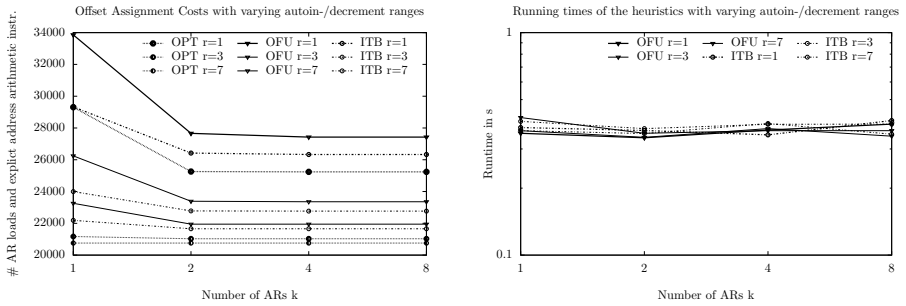

Figure 9: Offset assignment costs for all tested configurations and running times of the heuristics.

reasons why the instances become 'easier' or faster to solve for `GOA-AIP` with increasing r.

Fig. 9 shows the results of our experiments. The left image shows the impact of the various configurations for r and k on the total offset assignment cost, accumulated over all of the 1918 instances. The central observation is that the amount of address code can be considerably reduced when exploiting larger auto-modify ranges. At least for the evaluated instances, it appears that the reduction potential by increasing r is much higher than by increasing k since the offset assignment costs do not further decrease significantly for $k > 2$. However, this is partially also due to the already discussed character of the instances. The results approve the already in [6] observed impression, that the performance loss when using the proposed heuristics is rather small - also for increasing auto-modify ranges. The `GOA-ITB-MCF`-layout is already a better basis for the ARA part than an order-of-first-use layout. However, there is further potential in achieving near-optimal solutions by generating memory layouts that are not SOA-oriented but already take larger auto-modify ranges into account. Using an optimum address register assignment is worthwhile and computationally not too intensive so that it can be performed in production compilers. As can be seen in the right image, the heuristics are fast and sum up to less than half of a second in total although many small min-cost flow problem need to be solved.

References

[1] R. Leupers, Offset assignment showdown: Evaluation of DSP address code optimization algorithms, in: Proc. 12th Int. Conf. on Compiler Constr., CC'03, Springer, 2003, pp. 290–302.

[2] J. Huynh, J. N. Amaral, P. Berube, S. A. A. Touati, Evaluating address register assignment and offset assignment algorithms, ACM Trans. Embedded Comput. Syst. 10 (3) (2011) 37.

[3] M. Jünger, S. Mallach, Solving the simple offset assignment problem as a traveling salesman, in: Proc. 16th Int. W. on Softw. and Compilers for Embed. Syst., M-SCOPES '13, ACM, 2013, pp. 31–39.

[4] D. H. Bartley, Optimizing stack frame accesses for processors with restricted addressing modes, Softw. Pract. Exper. 22 (2) (1992) 101–110.

[5] S. Liao, Code generation and optimization for embedded digital signal processors, Ph.D. thesis (1996).

[6] S. Mallach, R. C. Lozano, Optimal general offset assignment, in: Proc. 17th Int. W. on Softw. and Compilers for Embed. Syst., SCOPES '14, ACM, New York, NY, USA, 2014, pp. 50–59. doi:10.1145/2609248.2609251.

[7] N. Sugino, S. Iimuro, A. Nishihara, N. Fujii, DSP code optimization utilizing memory addressing operation, IEICE Transactions on Fundam. of Electr., Commu. and Comp. Sc. 79 (8) (1996) 1217–1224.

[8] R. Leupers, P. Marwedel, Algorithms for address assignment in DSP code generation, in: Proc. 1996 IEEE/ACM Int. Conf. on Computer-Aided Design, ICCAD '96, IEEE Comput. Soc., 1996, pp. 109–112.

[9] R. Leupers, F. David, A uniform optimization technique for offset assignment problems, in: Proc. 11th Int. Symp. on Syst. Synth., ISSS '98, IEEE Comput. Soc., 1998, pp. 3–8.

[10] O. Ozturk, M. T. Kandemir, S. Tosun, An ILP based approach to address code generation for digital signal processors, in: G. Qu, Y. I. Ismail, N. Vijaykrishnan, H. Zhou (Eds.), ACM Great Lakes Symp. on VLSI, ACM, 2006, pp. 37–42.

[11] A. Rao, S. Pande, Storage assignment optimizations to generate compact and efficient code on embedded DSPs, SIGPLAN Not. 34 (5) (1999) 128–138.

[12] S. Atri, J. Ramanujam, M. T. Kandemir, Improving offset assignment on embedded processors using transformations, in: Proc. 7th Int. Conf. on High Perf. Comput., HiPC '00, Springer, 2000, pp. 367–374.

[13] Y. Choi, T. Kim, Address assignment combined with scheduling in DSP code generation, in: Proc. 39th Design Autom. Conf., DAC '02, ACM, 2002, pp. 225–230.

[14] C. H. Gebotys, DSP address optimization using a minimum cost circulation technique, in: Proc. 1997 IEEE/ACM Int. Conf. on Computer-Aided Design, ICCAD '97, 1997, pp. 100–103.

[15] C. H. Gebotys, A minimum-cost circulation approach to DSP address-code generation, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 18 (6) (1999) 726–741.

[16] E. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, 1976.

[17] G. B. Dantzig, D. R. Fulkerson, S. M. Johnson, Solution of a large-scale traveling-salesman problem, Operations Research 3 (1954) 393–410.

[18] R. Burkard, M. Dell'Amico, S. Martello, Assignment Problems, SIAM, Philadelphia, PA, USA, 2012.

[19] B. Wess, M. Gotschlich, Optimal dsp memory layout generation as a quadratic assignment problem, in: Proc. 1997 IEEE Intern. Symp. on Circuits and Systems ISCAS '97, Vol. 3, 1997, pp. 1712–1715. doi:10.1109/ISCAS.1997.621465.

[20] M. Padberg, G. Rinaldi, A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems, SIAM Rev. 33 (1) (1991) 60–100. doi:10.1137/1033004.

[21] M. W. Padberg, M. R. Rao, Odd minimum cut-sets and b-matchings, Mathematics of Operations Research 7 (1) (1982) 67–80.

[22] CPLEX optimization studio version 12.6, Reference manual, IBM ILOG (2013).

[23] B. Dezső, A. Jüttner, P. Kovács, LEMON - an open source C++ graph template library, Electron. Notes in Theor. Comp. Sc. 264 (5) (2011) 23–45.

[24] Z. Király, P. Kovács, Efficient implementations of minimum-cost flow algorithms, CoRR abs/1207.6381.